by Jeff Bachiochi (USA)

Application Communication with USB (Part 1)

The Enumeration Process Explained

Consumers understand how to use USB technology. Developers must understand how it works. This article series is an excellent starting point. Here you learn how a USB device goes through a number of states as it progresses through the enumeration process.

In the past, when I wanted to give my project user I/O through a PC connection, it was as easy as 8–N–1. UARTs were an easy interface. In fact, a lot of serial communication used an 8-bit data length with no parity and a single stop bit. You could pretty much rely on using this default and connection was assured. I liked using the serial and parallel ports. Many considered these divine in their simplicity. However there may be a little confusion with DB25s and DE9s, not to mention their s...e...x. I think we could have come up with a smaller connection form factor to keep laptop manufacturers happy, but users seemed to be shouting, "Make our lives easier!" The user drives the market. And so, we have the universal serial bus (USB).

For the user, it's (pretty much) plug and play. No matter what the peripheral—keyboard, mouse, modem, printer, camera, or bulk storage—it's all the same. Plug it in (a driver might have to be selected once) and then it just works, first time, every time—or so the dream perpetuates. Although USB continues to evolve, as a user, connecting peripherals has become a no-thought process, like tying a shoe lace. We plug it in and bing-bong, we're ready to go.

For a developer, it's a different story. In order for every peripheral to play through the standard USB connection, it must follow a rigorous enumeration process. Enumeration is the process of exchanging information that allows the host (connection manager) to learn about a connected device (peripheral). By asking the right questions, the host can determine which kind of device is attached and find the right driver to load for that device. There are many device drivers that come standard with your OS (assuming it supports USB). If a standard driver does not fit the device, the OS will prompt you to supply one. For more information on how a device driver is selected, I suggest you pick up a copy of Jan Axelson's *USB Complete*.

Every USB device will go through a number of states as it progresses through the enumeration process. It begins with an "unattached" state. Once a device is plugged into the USB

Transfer type	Control	Bulk	Interrupt	Isochronous
Typical use	Configuration	Printer (Delivery time not critical, but must get there without errors)	Mouse (Needs to be delivered in a timely fashion without errors)	Audio (Needs to be delivered on time, but can function with sporadic data errors)
Data	Message	Stream	Stream	Stream
Error correction	Yes	Yes	Yes	No
Guaranteed delivery rate	No	No	No	Yes
Guaranteed latency	No	No	Yes	Yes

Table 1—While a USB device could get by using just Control transfers, the Bulk, Interrupt, and Isochronous methods have been implemented to guarantee specific requirements.

Transfer type	Control			Bulk	Interrupt	Isochronous
Stage	Setup	Data (in or out)	Status	Data (in or out)	Data (in or out)	Data (in or out)
Phase	Token	Token	Token	Token	Token	Token
	Data	Data	Data	Data	Data	Data
	Handshake	Handshake	Handshake	Handshake	Handshake	

Table 2—The Control transfer is the only transfer method that consists of multiple Stages: Setup, Data (optional), and Status. Every stage requires token, data, and handshaking packets (except the Isochronous method). The handshaking phase is not required in an isochronous transfer since there is no error correction.

bus, it enters the "attached" state. The host can apply power to the device if necessary and the device enters the "powered" state. After retrieving some initial information from the device, the host assigns the device its own unique address, taking the place of its default address of 0. Now the device is in the "addressed state." Additional information retrieval takes place and if the process completes properly, the device is now in the "configured" state and ready for application communication. None of this can happen without data being passed. So let's start out with a look at the transfer process.

TRANSFERS

RS-232 communication consists of a single type of transfer, where an application passes data back and forth between devices at a prearranged protocol (normally 8 data bits, 1 stop bit, no parity). USB communication has two types of transfers: configuring and message. In order for this whole idea of using a single interface for every peripheral to work, the host must be able to somehow configure itself based on what gets plugged in. So, where the user was originally responsible for the configuration of the connection by setting up both UARTs to the same protocol, with USB this configuration is handled by the host without user intervention. USB has multiple conversations happening simultaneously. Not are only configuration and message conversations taking place between the host and an attached device, but potentially multiple devices, all over the same wires.

A device must know whether the configuration or application data being passed is for itself or another device. This is handled by the USB protocol. I've discussed the electrical side of USB in a past column (you can find a list of my past USB columns at the end of this article), so I won't discuss connection speed or packet timing here, but concentrate on the transfers themselves. It is sufficient to say that USB communication uses one of four transfer methods: control, bulk, interrupt, and isochronous. However, all USB communications must use the control transfer to make use of the predefined functions (in the USB specifications) and perform the enumeration process. Table 1 shows typical uses for these transfer methods.

The control transfer is the only transfer that can operate bidirectionally through its particular pipe or connection. While a pipe is not a physical connection, it is an association between the host and a device's endpoint. To start the enumeration process, each newly attached device must default to a control pipe with an endpoint of 0. The host will use this default address to communicate with this new device and begin the enumeration process by requesting information. To be able to share the bus with all devices, all information to be transferred is broken down into one or more transactions. Multiple transactions are necessary when all the information won't fit into a transaction's fixed capacity. A transaction can have multiple stages, most consisting of a token packet, a data packet, and a handshake packet (see Table 2). As a developer, our requirements have been simplified by the introduction

Register bits	7	6	5	4	3	2	1	0
UCON	—	Ping-Pong Reset	Single Ended Zero	Packet Transfer Disable	USB module enable	Resume Enable	USB Suspend	—
UCFG	Eye Pattern Enable	OE Monitor Enable	—	On-chip Pull-ups Enabled	On-chip Transceiver Disable	Full Speed Enable	PPB CFG1	PPB CFG2
USTAT	—	ENDP3	ENDP2	ENDP1	ENDP0	Transaction Direction	Transaction PP Bit	—
UADDR	D7	D6	D5	D4	D3	D2	D1	D0
UFRML	D7	D6	D5	D4	D3	D2	D1	D0
UFRMH	—	—	—	—	—	D10	D9	D8
UEP0	—	—	—	EndPoint Handshake Enable	Control EndPoint Disable	Output Endpoint Enable	Input Endpoint Enable	Endpoint Stall Indicator
...								
...								
UEP15	—	—	—	EndPoint Handshake Enable	Control EndPoint Disable	Output Endpoint Enable	Input Endpoint Enable	Endpoint Stall Indicator

Table 3—The first two registers configure the SIE. The USTAT register reflects the state of a four-deep FIFO keeping track of transactions. Each transaction will have an associated pipe or Endpoint 0:15 and will indicate the direction of the transaction Setup/Output or Input and a ping-pong bit (which I'll skip over for now). UADRR is an assignment for future communications by the host once the device has been enumerated through a default assignment of ADDR=0. The UFRMH:L register pair is used in isochronous transfers to monitor the present frame number. Finally, the last 16 registers, one for each possible endpoint, ENDP0:15, configure how the endpoint is used.

of some special hardware.

SIE

Today, many microcontrollers contain a USB peripheral that handles all of the USB bus interfacing. This support takes care of the hardware and its associated bus issues and leaves you to deal with the actual packets. I will be using the interrupt mode (as opposed to polling) for this project, so most of the work will take place in response to USB interrupts set by the serial interface engine (SIE). While this project uses a Microchip Technology PIC18F4450, and the information presented here may be specific to this manufacturer, the procedural outline should be similar for any micro depending on the implementation of its USB peripheral.

I've broken down the registers associated with the SIE into two groups: the control/status and the interrupt registers. The control/status registers are basically used to configure the USB SIE. I've listed them in Table 3 so you can refer to them while we discuss the code developed for this project. I've used a total of three endpoints. These include the default, control endpoint0, an input endpoint1 (configured for interrupt transfers), and both input and output endpoint2s (configured for bulk transfers).

The interrupt registers are divided into status and error interrupts (see Table 3). Most of the action taking place during enumeration will be from either the USB Reset or the Transaction Complete interrupts.

READY, SET, SAMPLE

Before getting into the actual code, let's consider the hardware setup I've been using. To monitor USB traffic between the host and the device, I used a protocol analyzer made by Total Phase called the Beagle USB480. This product can capture all of the USB events that pass through it. Using an analyzer helps you keep track of the communication packets and helps determine what your code is reacting to. Photo 1 is the Beagle's datacenter view of USB activity when I plug my project into the bus through the analyzer. Note the top two green entries. Here the host is resetting the bus and checking the attached hardware to determine what bus speed to use for communications. Then the enumeration process begins with the first transfer labeled "Get Device Descriptor."

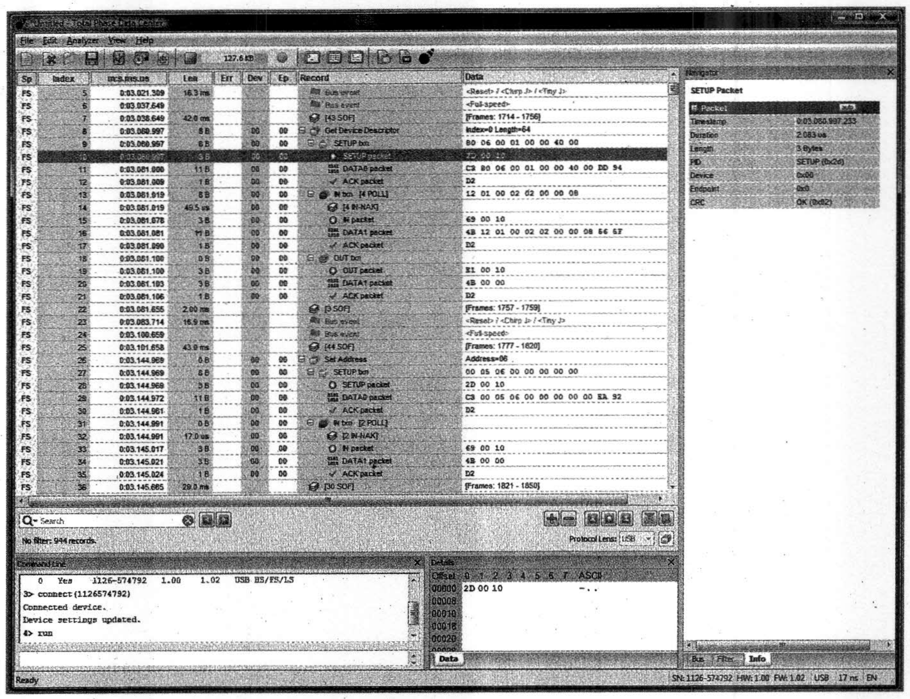


Photo 1—This view of the Total Phase Data Center application shows the logging of USB events that have been captured by the Beagle USB 480 protocol analyzer. This project has just been plugged into the USB bus and is beginning an enumeration sequence.

The enumeration process is like reading the device's datasheet. Enumeration gathers the parameters that describe the device, and it begins with the Standard Device Descriptor. To retrieve this information from the newly attached USB device, the host issues a control transfer. Refer to Figure 1 for the three stages of a control transfer. Note that each stage (setup, data, and status) of a transfer consists of three transaction phases (token, data, and handshake). Each transaction ends with the SIE setting the Transaction Complete interrupt.

Let's begin with that first transaction—the setup stage. Figure 2 shows

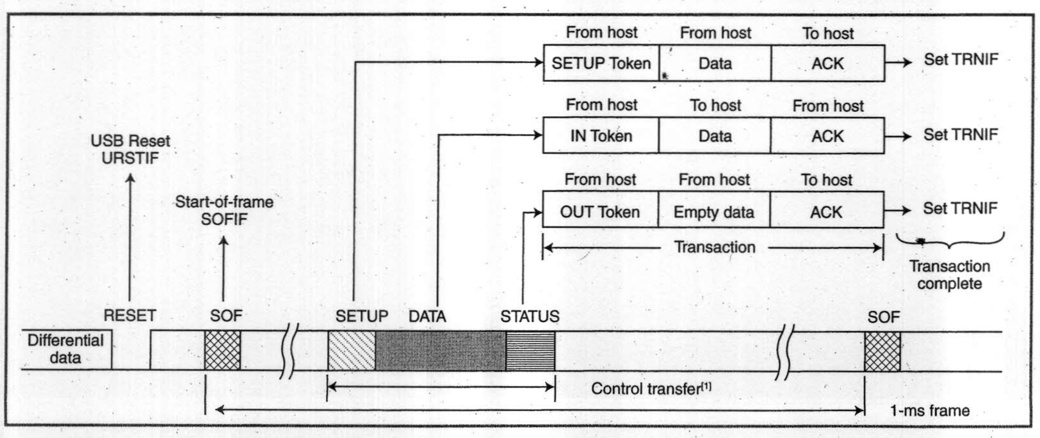


Figure 1—USB communications will consist of 1-ms frames that can contain many transactions between a host and connected devices. This figure shows a potential sequence of three transactions that will complete a control transfer within a single frame. The control transfer shown here is only an example showing events that can occur for every transaction. Typical control transfers will spread across multiple frames.

the packet protocol for each transaction phase on the USB bus. The SIE recognizes the first transaction as a setup token, receives the packet, and indicates when it is complete. The Transaction Complete interrupt is raised and it's up to us to do something with it. So where does this transaction end up?

SYNC	PID	ADDR	ENDP	CRC	EOP
8 bits (low/full) 32 bits (high)	8 bits	7 bits	4 bits	5 bits	N/A

SYNC	PID	DATA	CRC	EOP
8 bits (low/full) 32 bits (high)	8 bits	Up to 8 bytes (low)/1,023 bytes (full)/1,024 bytes (high)	5 to 16 bits	N/A

SYNC	PID	EOP
8 bits (low/full) 32 bits (high)	8 bits	N/A

Figure 2—Here are the three transactions that form a control setup transfer: setup, data, and acknowledgement. After a specially formatted sync byte, each packet begins with a PID byte. The PID (see Table 4) determines the data that follows.

DUAL-PORT RAM

Besides setting the Transaction Complete interrupt, the SIE has also placed an entry into a 4-byte FIFO called USTAT. This information came from the 3-byte setup token (highlighted in Photo 1). Referring to Figure 2, that's the PID, address, and endpoint from the host.

When you read USTAT, you are getting the first transaction received by the SIE (the next entry in the FIFO becomes available as soon as the Transaction Complete interrupt is cleared). USTAT holds the last endpoint received and direction of the transaction, in this case endpoint0 and setup/out (bit 7 of the PID). Table 4 shows the breakdown of the first byte of every transaction—the PID.

PID Type	PID Name	PID<3:0>*	Description
Token	OUT	0001B	Address + endpoint number in host-to-function transaction
	IN	1001B	Address + endpoint number in function-to-host transaction
	SOF	0101B	Start-of-frame marker and frame number
	SETUP	1101B	Address + endpoint number in host-to-function transaction for SETUP to a control pipe
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high-speed, high bandwidth isochronous transaction in a microframe
	MDATA	1111B	Data packet PID high-speed for split and high bandwidth isochronous transactions
Handshake	ACK	0010B	Receiver accepts error-free data packet
	NAK	1010B	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported
	NYET	0110B	No response yet from receiver
Special	PRE	1100B	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	ERR	1100B	(Handshake) Split Transaction Error Handshake (reuses PRE value)
	SPLIT	1000B	(Token) High-speed Split Transaction Token
	PING	0100B	(Token) High-speed flow control probe for a bulk/control endpoint
	Reserved	0000B	Reserved PID

Table 4—The PID value is made up of the lower nibble above and an upper nibble, which is simply the complement of the lower nibble, thus holding its own checksum. The 0x2D PID (Packet Indentifier Field in the highlighted transaction in Photo 1) indicates a setup token. The following two bytes hold address, endpoint, and CRC information (0x00 and 0x10 = 0b00000000 and 0b00010000 = address 0b0000000, endpoint 0b0000, and crc 0b1000) as in Figure 2.

The USTAT information refers to a special dual-port RAM area that the SIE shares with the micro, in this case 256 bytes in length (0x0400–0x04FF). This area is used for passing all of the data that moves through any established pipe, which at this point is only the default control pipe between the host and endpoint0. This area will eventually be used for two purposes: operational information and data.

Remember that unlike the single data channel for RS-232 communications, USB has configuration and messaging information that shares the same bus. Each transaction from the SIE uses a set of four registers, one set for each of the possible endpoints—EPIn0:15 and EPOut0:15. In fact, this could be 64 endpoints when using ping-pong buffers, but we won't discuss this option here. That is a potential use of 32 × 4, or 128 registers. That's half the allotted dual-port RAM. The remaining 128 bytes of dual-port RAM would be used as buffers for each of these endpoints. You can see that space is at a premium here. This project, as previously mentioned, will use a total of five endpoints. At this point, the host only knows about two: the default control pipe to endpoint0, which has an input endpoint (EP0In), and an output endpoint (EP0Out).

Part of the design process is to allocate 4 bytes (BDs or Buffer Descriptors) for each endpoint that will be

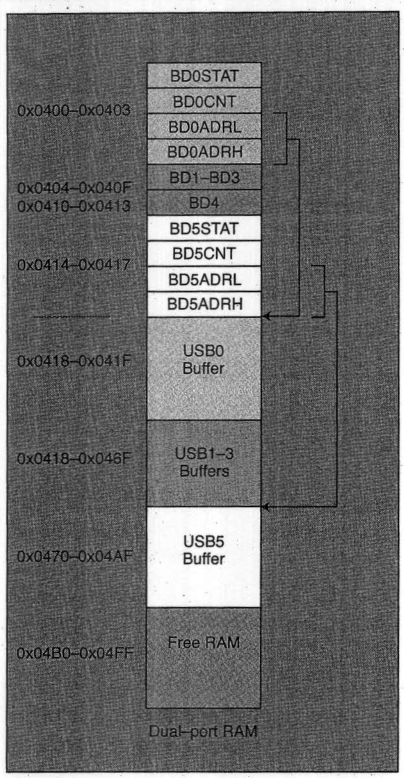


Figure 3—The four byte groups of Buffer Descriptors have predefined locations in the dual-port RAM. Their ADRH:L registers are used to define the actual data buffers used by each enabled endpoint. This project uses five of the first six buffers. BD4 (EP1In) is not used. All RAM after the last enabled BD can be used for the actual buffers. In this project, USB0-2 will be 8 bytes in length, and USB3 and USB5 will be 64 bytes in length. Note that the BD4 registers are accounted for and are free RAM; since it isn't enabled, no USB4 Buffer exists.

used in your application. Referring back to Table 3, UEP0:15 defined if and how each of the endpoints will be used. While the UEP0 register is the default control endpoint and will normally have bits D3:1 set, it's up to the designer to decide if and how the remaining endpoints will be used. The location of each endpoint has been predefined. Since the endpoint allocation is 4 bytes for each endpoint, the SIE knows where each set of 4 bytes begins. In this project, EP0Out is placed at the first byte (0x0400, BD0) of the dual-port RAM with EP0In following at 0x0404 (BD1), as you can see in Figure 3. Each byte of the BD has an important function.

Beginning with the last 2 bytes, BD*n*ADRH:L (where n = 0-32) form an address pointer to USB*n*B (USB*n* Buffer) where the endpoint data will be found. The size (length) of this buffer is defined during the design process. The size can vary by endpoint and is defined in the Descriptor Tables. (The Descriptor Tables are the device's datasheet; we'll get to these later on.)

The second byte, BD*n*CNT, indicates the actual count of pertinent data in the buffer. The first byte, BD*n*STAT, can be a little confusing as it has a multiple purpose. Since the dual-port RAM can be accessed by the SIE and the user application, bit 7 of BD*n*STAT is a semaphore flag used to indicate who owns the BD and USB*n*B. When BD*n*STAT.7 = 1, the SIE owns the endpoints' RAM and the user should not alter the endpoints' BDs or USB*n*B. The SIE will release these to the user by clearing BD*n*STAT.7 when it is finished, signalling that there is stuff there for the user. When you finish reading from or writing to the BDs and USB*n*B, you give it back to the SIE by setting BD*n*STAT.

The rest of the BD*n*STAT bits take on different meanings depending on who owns the BD's dual-port RAM. When the SIE hands you these buffers (BD*n*STAT.7 = 0), the BD*n*STAT holds the received token's PID (Packet IDentifier). With this value, you will know what to do with the data in the USB*n*B!

When you hand the BDs and USB*n*B back to the SIE, the BD*n*STAT holds some housekeeping information. The most important housekeeping item is the data toggle bit. As a means of making sure that stuff is passed back and forth between the host and the device correctly, data toggling is implemented. This method simply marks each packet using the data toggle bit that is flipped after each use. If a packet is missed, data toggling will be out of sync and the packet can be rejected.

GET ON WITH IT

The data we found in USTAT tells us that we are dealing with EP0Out. The associated BD for EP0Out is BD0. The four BD0 registers—BD0STAT, BD0CNT, BD0ADRL, and BD0ADRH—beginning at address 0x400 have all we need to know to access the first transaction. So assuming that the USB interrupts have been enabled via PIE2.5 along with peripheral and global enable (INTCON.7:6), execution has been tossed to the interrupt routine. This tests every USB interrupt in registers UIE and UIR (referring back to Table 5) to see which USB interrupts are enabled and which needs servicing. For now we are interested in only two: Transaction Complete and USB Reset. If you are clever you may have noticed (from Photo 1) that a USB Reset had occurred prior to the Get Device Descriptor request.

The host will want a device to interpret its request correctly, so it will begin by putting a device into a known state by requesting a reset. The USB Reset interrupt requests this, and our routine clears out the FIFO, UADDR, and UEP0:15 registers. It then sets up the BDs for the default control endpoint0 (BD0, EP0Out and BD1, EP0In), endpoint1, and endpoint2. This includes the USB0:5 buffer address pointers for each BD,

Interrupt register bits	7	6	5	4	3	2	1	0
Status UIR and UIE	—	Start of frame token	Stall handshake	Idle detect	Transaction complete	Bus activity detect	USB Error condition	USB Reset
Error UEIR and UEIE	Bit stuff	—	—	Bus turnaround timeout	Data field size	CRC16	CRC5	PID Check failure

Table 5—Any interrupt can be enabled via the *IE registers. Interrupt conditions are revealed via the *IR registers. Bit1 of the Status Register is a global Enable and Status bit of the Error registers.

the third and fourth bytes of each BD (BD*n*ADRH:L). The endpoint control UEP0 is configured for the default, a control endpoint using setup, in, and out transfers. UEP1 will be output (interrupt) transfers only and UEP2 will use both in and out (bulk) transfers. Finally, all of the USB interrupts are cleared and you exit the interrupt routine.

Following the flow of USB activity based on the Beagle's logging (back in Photo 1), execution has been directed again to the interrupt routine. This time it was the Get Device Descriptor request that set the Transfer Complete interrupt. You don't know what the request was yet, so you use the USTAT register to determine which BD to service. The BD's four bytes

Offset	Field	Size (bytes)	Value	Description
0	bmRequestType	1	Bitmap	Characteristics of request:
				D7: Data transfer direction
				0=Host-to-device
				1=Device-to-host
				D6..D5: Type
				00=Standard
				01=Class
				10=Vender
				11=Reserved
				D4..D0: Recipient
				00000=Device
				00001=Interface
				00010=Endpoint
				00011=Other
				001xx-111xx=Reserved
1	bRequest	1	Value	Specific request
				0=Get_Status
				1=Clear_Feature
				2=Reserved
				3=Set_Feature
				4=Reserved
				5=Set_Address
				6=Get_Descriptor
				7=Set_Descriptor
				8=Set_Configuration
				9=Get_Configuration
				10=Get_Interface
				11=Set_Interface
				12=Synch_Frame
2	wValue	2	Value	Word-sized field that varies according to the request, for descriptors
				0x0100=Device
				0x02xx=Configuration
				0x03xx=String
				0x0400=Interface
				0x0500=Endpoint
				0x0600=Device_Qualifier
				0x0700=Other_Speed_Configuration
				0x0800=Interface_Power
4	wIndex	2	Index or Offset	Word-sized field that varies according to the request. Typically used to pass an index or offset
6	wLength	2	Count	Number of bytes to transfer if there is a Data stage

Table 6—The USB Device Request might be the most informative group of information transferred via USB. The first byte completely describes how the request should be handled while the second byte defines the request.

plus the USTAT register are moved into a set of working registers to make them easier to interrogate. This Transfer Complete interrupt routine will be handling three of the four token PIDs possible in a control transfer, setup, in, and out (see Table 2 and Table 4). BDSTAT holds the PID that determines which of these to service. The value 0x2D tells us we need to service a setup token (actually it's the "D" that indicates the PID, the "2" is just D's complement).

We branch to the `ProcessSetupToken` routine to continue this interrupt service. BD0ADRH:L shows where to find the USB0Buffer, and BD0CNT indicates how many characters are significant. USB0Buffer holds the data phase of the setup stage. It should be a USB Device Request of eight bytes. These eight bytes in the USB0Buffer are moved to another set of working registers (so we won't need pointers anymore to access them). Table 6 shows the format of the USB Device Request. Referring back to Photo 1, the second byte in the data phase of the setup stage is GetDescriptor (0x06) and the host is requesting 0x40 characters. We're through processing the USB0Buffer, so the BD can be released to the SIE (BD0STAT.7 = 1). You then clear the packet-transfer disable that is set by the SIE when a setup token is received (UCON.PKTDIS = 0). This action signals the SIE that we've serviced it. Now we need to make a branch to handle the type of descriptor request. This is based on the first byte of the data phase (BMREQUEST.6:5), in this case it is a Standard Descriptor.

Within the Standard Descriptor branch there are 10 types of requests (offset 1 in Photo 1). We've already determined that we need to service the GetDescriptor, however there are eight possible descriptor types (offset 2 in Photo 1). Luckily, each request doesn't use all of the possible types. For this project, we need only three: device, configuration, and string descriptors. The upper byte of the wValue in the data phase indicates the descriptor type; in this case it is Device Descriptor (0x0100). The lower byte will be zero unless there are multiple configurations or strings, in which case this will indicate an array index.

We now know that the host is requesting a GetDescriptor transfer and that we need to send the Standard Device Descriptor. This is the first item in a device's datasheet. Next month, we'll look at this and the other items that make up the datasheet (or USB Descriptors) for this project. Keep this column handy, as there is much to refer back to next time. I encourage you download a copy of the "Universal Serial Bus Revision 2.0" specification. Of particular importance is Chapter 9 as this is the essence of communication and enumeration. In addition, download the "Class Definitions for Communication Devices 1.2." USB devices have been grouped by function into classes. Since this project's USB device is of the communication class, you might wish to look through the requests and notifications that are specific to this kind of device. I'm hopeful this project will help you to wade through the USB specifications and locate those parts that are meaningful to your particular design. ■

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. You can reach him at jeff.bachiochi@imaginethatnow.com or at www.imaginethatnow.com.

RESOURCES

J. Axelson, *USB Complete: Everything You Need to Develop Custom USB Peripherals*, 2nd ed., Lakeview Research, Madison, WI, 2001.

J. Bachiochi, "Accessing the USB Framework," *Circuit Cellar* 195, 2006.

———, "Create a USB Hybrid Hub," *Circuit Cellar* 170, 2004.

———, "Embedded USB Breakthrough," *Circuit Cellar* 200, 2007.

———, "USB DMX," *Circuit Cellar* 172, 2004.

———, "USB in Embedded Design Part 1: The Undeniable Benefits," *Circuit Cellar* 165, 2004.

———, "USB in Embedded Design Part 2: HIDmaker Converts an Application," *Circuit Cellar* 165, 2004.

Class Definitions for Communication Devices 1.2, www.usb.org/developers/devclass_docs/CDC1.2_WMC1.1.zip.

USB Implementers Forum, www.usb.org.

USB Revision 2.0 Specification, www.usb.org/developers/docs/usb_20_122909-2.zip.

SOURCES

PIC18F4450 Microcontroller
Microchip Technology, Inc. | www.microchip.com

Beagle USB 480 Protocol analyzer
Total Phase, Inc. | www.totalphase.com

by Jeff Bachiochi (USA)

Application Communication with USB (Part 2)

The Importance of Descriptors

You know how a USB device goes through a number of states as it progresses during the enumeration process. The next topic to understand is the standard device descriptor, which is the first item in a device's datasheet.

I must be the reincarnation of a cat. I'm curious about many things. In fact, I've used a number of my "nine lives" by letting my curiosity take precedence over the self-preservation instinct. Do you know anyone who can say they've never performed an act of stupidity at some point in their life? Somewhere along the line, we learn how fragile our bodies are and that tempting fate should not be an everyday experience. At this point, curiosity takes a backseat to personal safely. We are more apt, say, to learn about aerodynamics through references and modeling, rather than jumping off a cliff with a hang glider.

I've been interested in USB since it was forced upon us years ago. Even though many manufacturers have given us the tools to use it, without having to know how it really works, I just can't leave it at that. I want to know more about its operation. For those of you who might have similar feelings, I hope I can shed a little light on this by going through a simple USB project that will enumerate and transfer application data using assembly language. This project began last month. I encourage you to start there and get up to speed on the USB hardware peripheral—the serial interface engine (SIE) built into many of Microchip Technology PIC microcontrollers. While each manufacturer may arrange things differently, they all must obey the same rules. Thus, you should be able to take what you learn here and apply it to any micro.

We left off looking at the start of the enumeration process as seen from the Beagle USB 480 protocol analyzer I use to display the bus activity between the USB host (my PC) and the project's device circuitry. After forcing a USB reset, the host has requested a descriptor. Through the SETUP packet received by the SIE, I discovered that the host was requesting a standard device descriptor (SDD) via a `GetDescriptor` transfer.

We know that the purpose of designing the USB interface was to make it user-friendly yet universal. That means that the host must be able to figure out how to access a device automatically when a user plugs it into the bus. To do this, a device must be able to answer all the questions the host will ask it. These questions have been predetermined by the USB specifications. So, as a device, we need to be ready to respond with the answers. The answers come from USB descriptors, a collection of information describing the device. Each descriptor uses an identical format to simplify the exchange of information. A descriptor begins with a length byte followed by a type byte. Let's begin with the descriptors, as defined in this project, to describe its configuration to the host, the

Standard Device Descriptor					
Offset	Field	Size	Format	Description	Project Value
0	bLength	1	Number	Descriptor length in bytes	0x12
1	bDescriptorType	1	Constant	DEVICE	0x01
2	bcdUSB	2	BCD	USB Specification revision	0x0200
4	bDeviceClass	2	Class	Class Code	0x02
5	bDeviceSubClass	2	SubClass	Sub Class Code	0x00
6	bDeviceProtocol	1	Protocol	Protocol Code	0x00
7	bMaxPacketSize0	1	Number	Maximum packet size EP0	0x08
8	idVendor	1	ID	Vendor ID	0x04D8
10	idProduct	1	ID	Product ID	0xFE96
12	bcdDevice	1	BCD	Device Revision	0x0000
14	iManufacturer	1	Index	Manufacturer String Index	0x01
15	iProduct	2	Index	Product String Index	0x02
16	iSerialNumber	1	Index	Serial Number String Index	0x00
17	bNumConfigurations	1	Number	Maximum Device Configurations	0x01

Table 1—The standard device descriptor describes general information about the device. It indicates to the host how many configurations it should look for that are associated with the device.

device descriptor, the configuration descriptor, and the string descriptor.

SDD

A USB device has only one SDD as it is a global description of the device and all its configurations. Table 1 shows that this descriptor contains the version number of the USB specifications that it conforms to, so the host will understand the rules it can play by.

Next are the device's class and subclass. These define the kind of device—in our case, it is the communications device class, covering telecommunications and networking devices. The communication device class has no subclass or device protocol, so these bytes are zero. The packet size byte tells the host a packet's maximum size. For EndPoint0, that's 8, 16, 32, or 64 bytes.

Next, the device descriptor lists the vendor ID (that's Microchip Technology), the product ID (that's mine), and a release or version number for my application. Because all this is kind of cryptic, the descriptor includes a number of bytes that can point to string descriptors that will describe the manufacturer, the product, and the serial number using ASCII text. These may show up in various places on your PC when installing drivers or viewing system information. Finally, one last byte indicates the number of configurations your device supports. In this project, we have just one.

My SDD is 0x12 bytes long, so how can the host get this if I'm using a maximum packet size of eight? If you refer back to the first part of this series, you'll see that the host follows its `GetDescriptor` set-up transaction with additional "in" token requests until it has received all it requires. But hold on to that thought; we haven't discussed the "in" and "out" token yet.

SCD

Unlike the SDD, there may be more than one standard configuration descriptor (SCD). Remember, the number of configurations was defined back in the last byte of the SDD. In this case, we're just dealing with a single configuration; therefore, we need only one configuration descriptor. The configuration descriptor is very complex. It is a container, if you will, for all the interface descriptors associated with this configuration. It uses the standard descriptor format so the host knows the descriptor's type and size by the first 2 bytes (see the table posted on the *Circuit Cellar* FTP site).

The SCD's next 2 bytes define the total size of all the descriptors associated with this configuration. Following this is the actual number of interfaces supported by this configuration, the number of this configuration, and the index of any string descriptor describing this interface. The last 2 bytes define the attributes of the configuration (self-powered or supports remote wakeup) and the maximum power consumption (if it were to require power through the USB connector).

In this project, the host discovers that this configuration (1) has two interfaces associated with it. It also finds that the device is self-powered and supports remote wakeup. If its local power is removed, it would require 100 mA. While the host discovered this configuration descriptor's length is only 9 bytes, it knew to look for a total length value within this descriptor. It can continue to read in additional transactions (using the "in" token) until it has all of this information. The host can then break down this data into additional descriptors looking for more information on each of the two interfaces.

Appended to, and as part of, the total SCD are any additional descriptors that are required to fully define the device's configuration. Here we have two interfaces to define, and each will have its own interface descriptor. The first interface (interface descriptor 0) has no alternative setting—so its value is zero (the first and only)—and the number of endpoints used for the interface is one. Also within the interface descriptor are communication interface class (2), subclass (2), and protocol (1) values. In this descriptor, the host learns that interface 0 is of the communication class using the abstract control model having one endpoint.

We know that USB devices require specific device drivers to be loaded on the PC. To support all possible USB devices, this driver would be mammoth. So, the driver that gets loaded for each device contains only the subset necessary for its associated device. The driver gives the host the ability to know what descriptors to find for that particular device. The subclass 2

Standard String Descriptor0					
Offset	**Field**	**Size**	**Format**	**Description**	**Project Value**
0	bLength	1	Number	Descriptor length in bytes	0x04
1	bDesriptorType	1	Constant	STRING	0x03
2	wLANGID	2	Number	Country Code	0x0409
Standard String Descriptor1					
Offset	**Field**	**Size**	**Format**	**Description**	**Project Value**
0	bLength	1	Number	Descriptor length in bytes	0x36
1	bDesriptorType	1	Constant	STRING	0x03
2	bString	N	Number	Text	"Microchip Technology, Inc."
Standard String Descriptor2					
Offset	**Field**	**Size**	**Format**	**Description**	**Project Value**
0	bLength	1	Number	Descriptor length in bytes	0x4C
1	bDesriptorType	1	Constant	STRING	0x03
2	bString	N	Number	Text	"Imagine That (www.imaginethatnow.com)"

Table 2—The standard string descriptors have two formats. String0 defines the language used by the host to talk to the user. The remaining strings contain ASCII text presented in a UNICODE presented string.

(abstract control model) has its own set of descriptors that are used to help describe its functions through an interface. There are 25 subtype descriptors for the communications class. We will be using four for this project: header, call management (CM), abstract control management (ACM), and the union functional descriptor. You can read about all of them in the communication class specifications. The header descriptor indicates the version of the specifications that the interface is adhering to. CM defines the call-handling capabilities—and there are none, in this case. The ACM identifies the capabilities of the interface. These include: network connection support; sending break, line, and serial states; and COMM features. Finally, the union descriptor defines the master interface (interface 0 in this case) and the subordinate interfaces. In this instance, there is only one other interface—interface 1.

Following the subclass descriptors for this interface, you must include the endpoint descriptors for this interface. One endpoint is for this interface and its descriptor identifies the actual endpoint (EP1In) using an interrupt transfer type. This short descriptor also includes the packet size (64 bytes) and how often the host will poll for data (2 ms).

The second interface (interface descriptor 1) has no alternative setting, so its value is zero (the first and only) and the number of endpoints used for the interface (two endpoints here). The data interface class (10) has no subclass and uses no protocol, so these values are zero. In this descriptor, the host learns that interface 1 is of the data class using no specific protocol and having two endpoints. This interface has no additional descriptors other than the two specified endpoint descriptors. One endpoint is defined as using endpoint (EP2Out) using a bulk transfer type. The second endpoint is defined as using endpoint (EP2In) using a bulk transfer type. Both endpoints will use a packet size of 64 bytes and request polls whenever there is bandwidth.

SSD

If any optional standard string descriptors (SSD) are used, the first descriptor must identify the language or languages (see Table 2). The 16-bit LANGIDs2 defines how the host should present text to user (i.e., "Hit a key to continue"). The remaining strings can give the user an ASCII text description of some cryptic value within the associated descriptor.

In this project, the second SSD (1) is associated with the Manufacturer ID in the SDD and the text reads "Microchip Technology, Inc." The third SSD (2) is associated with the product ID in the SDD and the text reads "Imagine That (www.imaginethatnow.com)."

RETRIEVE DESCRIPTORS

I reprinted the first page of transactions as captured by the USB 480 Protocol Analyzer in Photo 1 because it demonstrates how the host interacts with this device when plugged into the USB and enumeration begins. We left off last month with the SIE setting the transaction complete interrupt after receiving the set-up token via EP0. The buffer descriptor (BD0) associated with default endpoint (EP0Out) pointed us to the 8-byte EP0Out buffer that held the packet data received by the SIE. Upon moving and interrogating the data, it was found to be requesting 64 characters via a GetDescriptor request (0x80). The descriptor data we discussed above has been placed at the top of the program memory space by the assembler, so we know where it can be found.

You must prepare an 8-byte packet response for the SIE before clearing the interrupt. Once you clear the interrupt, the SIE will acknowledge the SETUP packet and the host will send an IN packet to get our data. You best be ready. To get the requested information, USB_desc_ptr is preloaded with the offset address of the character we want from the descriptor table. A call to the Descriptor routine loads the address of the beginning of the descriptor table into a table pointer that will fetch characters

from the program memory space. The offset is added to the table pointer, a character is fetched from that address, and returned. The character can be then transferred to the EP0In buffer and the next character sought. When EpoIn buffer is filled we can leave the interrupt routine and clear the interrupt, the SIE acknowledges the host.

Refer to Photo 1a. The host has continued with an IN token. The SIE will again set the transaction complete interrupt after sending the data we preloaded into EP0In buffer. Note that the 8-byte DATA1 packet of the IN token has the first eight characters of the SDD. This time the interrupt routine will discover an IN token request from the USTAT and branch to the `ProcessInToken` routine. Since I stored the SETUP token's GetDescriptor request in USB_dev_req, I know how I got here and must continue to preload 8-byte packets of data into the EP0In buffer until all the requested data has been sent, I run out of data to send, or the request is cancelled by a new SETUP token.

As you can see in Photo 1a, notice that after the host acknowledged the receipt of the first IN data, it sent an OUT token. This is an acknowledgment that the transfer request is complete. It decided it has what it wants. Not only did it find out what the maximum packet size was (0x08), but also that the SDD is 0x12 bytes in length (no sense asking for more than that).

Note that this GetDeviceDescriptor transfer contained three stages: a setup transaction (in this case one), a data transaction, and a status transaction. Each transaction stage has a token phase, a data phase, and a handshake phase (see Figure 1).

ADDRESS STATE

Up to this point, the communication has taken place on the default address UADDR=0. The host is now ready to assign the device a real address. The host sends a SETUP token. The receipt of the token by the device's SIE causes another transaction complete interrupt. USTAT reveals it was via EP0Out. By processing the EP0Out Buffer, we find the 8-byte packet contains a SET_ADDRESS request (0x00) and the new address is 0x06. While you might be tempted to change UADDR immediately, hold on. Since this is only the SETUP stage of this transfer, we must continue using the old address until the transfer is complete. This is the only time when you need to finish things before making use of the data received. Since this request has no DATA stage, we just preload a zero-length packet into the EP0In Buffer and it is used as the HANDSHAKE stage. We exit the interrupt routine, clearing the interrupt and the SIE ACK is the SETUP token.

The host sends an IN token (the beginning of the HANDSHAKE stage), the SIE receives the request and sends the EP0IN Buffer, and sets the transaction complete interrupt. USTAT reveals it was via EP0In. Since we stored the SETUP token's SET_ADDRESS request in USB_dev_req, we know how we got here. There is nothing expected as this is the end of the SetAddress request and all transactions are complete. We are now clear to change UADDR to 0x06. And, don't forget to reset our USB_dev_req to 0xFF (NO_REQUEST). Once we leave the interrupt routine, clearing the interrupt flag, we have transitioned into the ADDRESS_STATE.

CONFIGURE STATE

Note that the next SETUP token (and all subsequent tokens) are sent to the new address 0x06. Now the device pays attention only to those tokens containing the new address. This new SETUP token is again requesting the

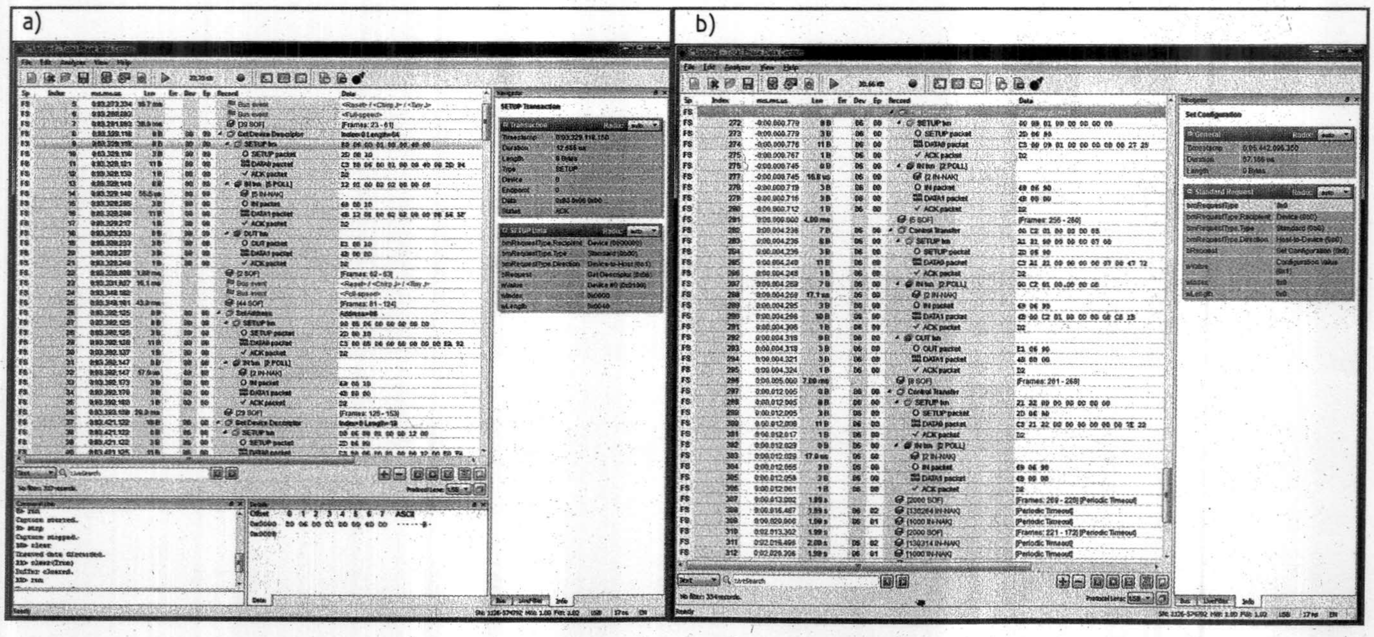
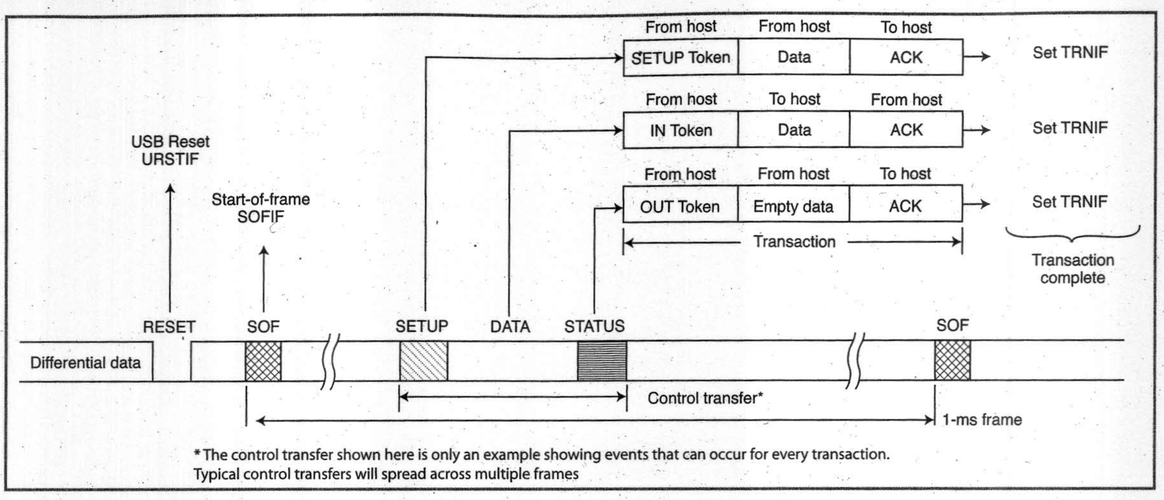


Photo 1a—The Beagle protocol analyzer's capture of the USB traffic clearly shows the transfer requests of a device beginning the enumeration process. Every device begins with a default address of 0 and is reassigned a new address by the SetAddress request seen here. b—The enumeration process ends when the host requests that a device be configured to one of its declared configurations. This project has a single configuration. Once configured, the host will begin polling for information according to the Endpoint Descriptors, as seen at the bottom here as IN_NAKs to EP2 and EP1.

Figure 1—This is a potential sequence of three transactions that will complete a control transfer. Each transaction or stage consists of a Token, Data, and Acknowledgement Phase or Packet.



GetDeviceDescriptor. This time it is only asking for 0x12 bytes. The host will continue to gain knowledge about the device by sending GetConfigurationDescriptor and GetStringDescriptor requests. Once it knows everything, the host will request a SetConfiguration and finally GetLineCoding and SetControlLineState via the class-specific requests.

The SetConfiguration request gives the device permission to use a specific configuration. In this case, we have defined only one. The configuration uses two interfaces and now we set them up using EP1In as an ACM interface using interrupt transfers (the control interface) and EP2In and EP2Out as the data interface using bulk transfers. The class-specific GetLineCoding request informs the host that our (virtual) UART is set to receive 8 data bits, with 1 stop bit, and no parity at a data rate of 115,200 bps. These are values I chose and stored in initialized variables so that they can be recalled and changed if necessary. I could set up the microcontroller's (physical) UART and this project would be a USB-to-UART interface, but that is an application left up to you. The SetControlLineState request presents the device with RTS and DTR states, which are controlled by the host.

The device is now in the CONFIGURE_STATE. It will be listed as a (virtual) COMM port in any application running on the PC that can use a serial port for data communication. The host starts to poll both interfaces of the device. It is polling EP1In looking for any control information that the device might want to convey to the host. It is also looking for any data on EP2In that the device may want to transfer to the host. The device is happy with the status quo and has no information to pass on, so the SIE has not been given control over any of the IN endpoints. It will NAK all polls, which tells the host that it's busy and to try again later. By "later" I mean (as defined in the standard endpoint descriptor) every two frames (2 ms) for the interrupt endpoint and whenever there is room for the bulk endpoint.

When an application on the PC selects this COMM port for use, things begin to happen once again. In my next article, I'll look at the two conversations that take place through the configured device's newly established endpoints, EP1 and EP2. ■

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. You can reach him at jeff.bachiochi@imaginethatnow.com or at www.imaginethatnow.com.

PROJECT FILES

To download the descriptor table, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2010/240.

SOURCES

PIC18F4450 Microcontroller
Microchip Technology, Inc. | www.microchip.com

Beagle USB 480 Protocol analyzer
Total Phase, Inc. | www.totalphase.com

NEED-TO-KNOW INFO

Knowledge is power. In the computer applications industry, informed engineers and programmers don't just survive, they *thrive* and *excel*.

For more need-to-know information about topics covered in Jeff Bachiochi's Issue 240 article, the *Circuit Cellar* staff recommends the following content:

Create a Hybrid Hub
by Jeff Bachiochi
***Circuit Cellar* 170, 2004**
Jeff shows you how to build a four-port USB hub with a TPS2071 power controller, a data controller, and a few other simple parts. Topics: USB, Hub, Power

Go to: www.circuitcellar.com/magazine/170toc.htm

Get Started with PIC USB Connectivity
by Jeff Bachiochi
***Circuit Cellar* 219, 2008**
Get started with USB. Jeff presents MCUs with USB peripheral support, embedded hosting, and more. Topics: USB, OTG, HNP Protocol, TPL

Go to: www.circuitcellar.com/magazine/219.html